DLP Lab4 Report

Shu Kai, Lin

April 2025

1 Introduction

In this lab, I implemented a conditional Variational Autoencoder (VAE) for video prediction tasks. The model is trained on 23,410 frames using paired pose and image data and is evaluated on 5 video sequences, each consisting of one initial frame and 630 pose labels. The implemented model predicts future frames based on pose sequences using a VAE framework combined with temporal modeling strategies. To improve performance and training stability, I experimented with multiple training hyperparameters, including different optimizers, KL annealing schedules, teacher forcing strategies, and reparameterization techniques. Detailed implementation choices, training methodologies, and analysis of results are presented in the following sections.

2 Implementation Details

2.1 Training/Testing protocol

2.1.1 Training

```
    def training_one_step(self, imgs, labels, adapt_TeacherForcing):
        # B, T, C, H, W = img.shape
        imgs = imgs.permute(1, 0, 2, 3, 4)  # (T, B, C, H, W)
        labels = labels.permute(1, 0, 2, 3, 4)
        losses = 0.0
        mse_losses, kl_losses = 0.0,0.0
        last_img = imgs[0]
        for t in range(1,self.train_vi_len):
            cur_img = imgs[t]
            if adapt_TeacherForcing:
                last_img = imgs[t-1]
            cur_img = self.frame_transformation(cur_img)
            last_img = self.frame_transformation(last_img)
            label = self.label_transformation(labels[t])
            z, mu, logvar = self.Gaussian_Predictor(cur_img,label)
            output = self.Decoder_Fusion(last_img,label,z)
            pred = self.Generator(output)
            pred = torch.clamp(pred, 0.0, 1.0)
            last_img = pred.detach()
            mse_losses += self.mse_criterion(pred,imgs[t])
```

```python
            kl_losses += kl_criterion(mu,logvar,self.batch_size)
        self.optim.zero_grad()
        losses = mse_losses + self.kl_annealing.get_beta() * kl_losses
        losses.backward()
        self.optimizer_step()
        return losses
```

The training_one_step function performs a single training iteration and model forwarding. It starts by rearranging the input image and label tensors so that the time dimension comes first, making it easier to loop over each frame in the sequence. For each time step starting from t = 1, the function optionally applies teacher forcing, which means using the ground-truth previous frame instead of the model's last prediction as input. Both the current frame and previous frame are transformed using preprocessing functions, and the corresponding label is also transformed. These are then passed to a Gaussian predictor module that produces the mean and log variance of a latent distribution, from which a latent vector z is sampled using the reparameterization trick. This latent vector, along with the previous frame and label, is passed into a decoder fusion module and then a generator to produce the predicted frame. The prediction is clamped to the valid pixel range and used as the next input if teacher forcing is not used. The mean squared error between the prediction and ground-truth frame is accumulated, as well as the KL divergence loss between the predicted distribution and a standard normal prior. After all frames are processed, the total loss is calculated by summing the reconstruction loss and the weighted KL loss, where the weight is controlled by a KL annealing schedule. Finally, gradients are computed and applied to update model parameters.

2.1.2 Validating

```python
    def val_one_step(self, imgs, labels):
        imgs = imgs.permute(1, 0, 2, 3, 4)  # (T, B, C, H, W)
        labels = labels.permute(1, 0, 2, 3, 4)
        with torch.no_grad():
            mse_losses = 0.0
            psnrs = 0.0
            last_img = imgs[0]
            for t in range(1,self.val_vi_len):
                cur_img = imgs[t]
                cur_img = self.frame_transformation(cur_img)
                last_img = self.frame_transformation(last_img)
                label = self.label_transformation(labels[t])
                z, mu, logvar = self.Gaussian_Predictor(cur_img,label)
                z = mu
                output = self.Decoder_Fusion(last_img,label,z)
                output[output != output] = 0.5
                pred =self.Generator(output)
                pred = torch.clamp(pred, 0.0, 1.0)
                last_img = pred.detach()
                mse_losses += self.mse_criterion(pred,imgs[t])
                psnrs += Generate_PSNR(pred,imgs[t])
        return mse_losses, psnrs / self.val_vi_len
```

The val_one_step function performs a forward-only validation pass to evaluate the model's video frame prediction quality. It begins by reordering the input tensors to make the time dimension first, enabling sequential frame-wise processing. Within a no-gradient context, it iterates through each time step from t = 1, applying preprocessing to the current frame, the previous frame, and the corresponding pose label. Instead of sampling from the latent distribution, it uses the latent mean (mu) directly as the input to the decoder. This choice ensures deterministic and consistent outputs during validation, allowing for stable performance evaluation without the randomness introduced during training. The decoder fusion module combines the previous frame, label, and latent mean to produce an intermediate output, which is then passed to the generator to reconstruct the predicted frame. Any NaN values in the decoder output are replaced with 0.5 to prevent numerical errors and visual artifacts—0.5 serves as a neutral midpoint in the normalized pixel range [0, 1]. The predicted frame is clamped to valid pixel bounds and used as input for the next step. Throughout the sequence, the function accumulates reconstruction loss (MSE) and peak signal-to-noise ratio (PSNR). At the end, it returns the total MSE and the average PSNR, providing a robust measure of the model's prediction accuracy over time.

2.1.3 Testing

```
    decoded_frame_list = [img[0].cpu()]
    label_list = []
    img_in = img[0]
    for i in range(label.shape[0]-1):
        img_in = self.frame_transformation(img_in)
        label_in = self.label_transformation(label[i])
        z, mu, logvar = self.Gaussian_Predictor(img_in,label_in)
        z = mu
        output = self.Decoder_Fusion(img_in,label_in,z)
        output[output != output] = 0.5
        pred = self.Generator(output)
        pred = torch.clamp(pred, 0.0, 1.0)
        img_in = pred
        decoded_frame_list.append(pred.cpu())
        label_list.append(label[i].cpu())
```

This testing loop generates future frames sequentially starting from the first input frame. It initializes the input image and iterates through each pose label. At each step, the current image and label are transformed and passed into the Gaussian predictor, where only the latent mean (‘mu‘) is used to keep predictions deterministic. The decoder fusion and generator produce the next predicted frame, replacing any NaNs in the intermediate output with 0.5 and clamping the final image to the valid pixel range [0, 1]. The predicted frame is then used as input for the next step. All predicted frames and labels are stored for later evaluation or visualization.

2.2 Reparameterization

```
    def reparameterize(self, mu, logvar):
        logvar = torch.clamp(logvar, min=-20, max=20)
        std = torch.exp(0.5 * logvar)  # exp still safe now
        eps = torch.randn_like(std)
        z = mu + eps * std
        return z
```

The reparameterize function is a core component of Variational Autoencoders (VAEs), enabling gradient-based optimization over stochastic latent variables. Given the mean (mu) and log variance (logvar) output by the encoder, the function clamps logvar to a safe numeric range to prevent instability. It then computes the standard deviation and samples noise eps from a standard normal distribution. The latent variable z is obtained via z = mu + eps * std, implementing the reparameterization trick, which shifts the sampling operation outside the computational graph, thus maintaining differentiability with respect to mu and logvar. This sampled z lies in the model's latent space — a lower-dimensional, continuous representation where complex patterns in the input data (e.g., video frames) are captured in a compact, probabilistic form. The latent space plays a vital role in controlling the generation and diversity of outputs. Ideally, similar inputs should map to nearby regions in latent space, enabling smooth interpolation, while sampling from the latent prior (e.g., a standard Gaussian) allows the model to generate coherent and plausible outputs. In conditional video prediction, the latent space encodes not only visual content but also temporal uncertainty, supporting the synthesis of realistic and temporally consistent future frames.

2.2.1 Teacher forcing strategy

```
    def teacher_forcing_ratio_update(self):
        if self.current_epoch > self.tfr_sde:
            self.tfr -= self.tfr_d_step
            self.tfr = max(0.0, self.tfr)
```

The teacher_forcing_ratio_update function gradually decreases the teacher forcing ratio (self.tfr) during training. It begins this decay only after a specified starting epoch (self.tfr_sde) is reached. Once past this point, the ratio is reduced by a fixed decrement (self.tfr_d_step) each epoch, ensuring a gradual transition from using ground-truth inputs to relying on the model's own predictions during sequence generation. To prevent the ratio from becoming negative, it is clamped to a minimum of 0.0. This approach helps the model adapt more smoothly to inference conditions and improves its generalization ability over time.

2.2.2 Kl annealing ratio

```
class kl_annealing():
    def __init__(self, args, current_epoch=0):
        self.current_epoch = current_epoch
        self.tol_epoch = args.num_epoch
        self.kl_type = args.kl_anneal_type
        self.kl_ratio = args.kl_anneal_ratio
        self.kl_cycle = args.kl_anneal_cycle
        if self.kl_type == "None":
            self.beta = 1.0
        else:
            self.beta = 0.0
    def update(self):
        self.current_epoch += 1
        if self.kl_type == "Cyclical":
            self.beta = self.frange_cycle_linear(
                step = self.current_epoch,
                n_iter=self.tol_epoch,
                start=0.0,
                stop=1.0,
                n_cycle=self.kl_cycle,
                ratio=self.kl_ratio
            )
        elif self.kl_type == "Monotonic":
            self.beta = min(1.0, self.current_epoch / (self.kl_ratio * 10))
        elif self.kl_type == "None":
            self.beta = 1.0
        else:
            raise ValueError("No such kl annealing type")
    def get_beta(self):
        return self.beta
    def frange_cycle_linear(self,step, n_iter, start=0.0, stop=1.0,  n_cycle=1, ratio=1):
        period = n_iter / n_cycle
        cycle_num = int(step / period)
        cycle_pos = step - cycle_num * period
        ramp = period * ratio
        if cycle_pos <= ramp:
            return start + (stop - start) * (cycle_pos / ramp)
        else:
            return stop
```

The kl_annealing class manages how the KL divergence weight, called beta, changes during training using one of three strategies: cyclical, monotonic, or none. In the cyclical strategy, beta increases linearly from 0 to 1 over a defined portion of each cycle, then stays at 1 for the remainder of that cycle, repeating this pattern for a set number of cycles. In the monotonic strategy, beta grows linearly from 0 to 1 over a fixed range of epochs and remains at 1 afterward. The none strategy disables annealing and keeps beta fixed at 1 throughout training. These strategies control how strongly the model is penalized for deviating from the prior in latent space, which helps balance reconstruction and regularization. The update method adjusts beta at each epoch based on the chosen strategy, and the current value can be retrieved with get_beta. Cyclical annealing encourages periodic exploration of the latent space, while monotonic annealing gradually introduces regularization pressure, making it easier to stabilize training.

3 Analysis & Discussion

3.1 Plot Teacher forcing ratio

3.1.1 Analysis & Compare with the loss curve

Below experiments are conducted with batch size = 12, learning rate = 5e-4, optimizer = AdamW, KL Divergence annealing type = Cyclical, number of epochs = 100, tfr_sde = 10, tfr_d_step = 0.1, and different teacher forcing ratio.

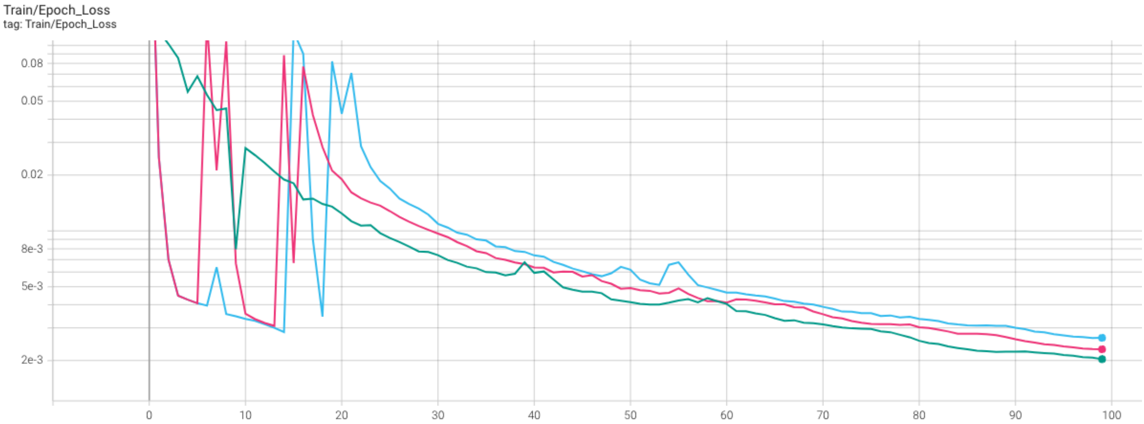
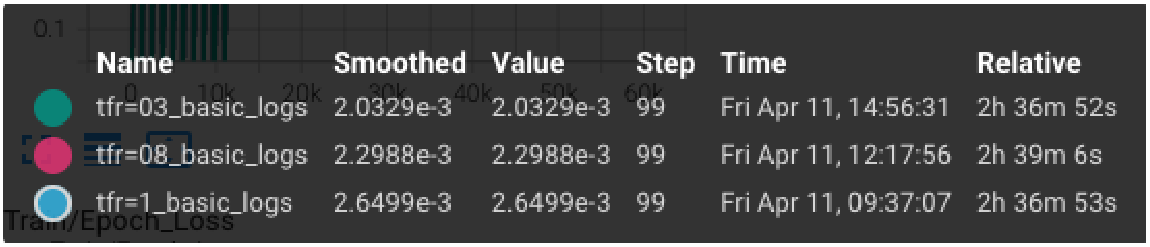


Figure 1: Teacher forcing ratio loss comparison



	Name	Smoothed	Value	Step	Time	Relative
●	tfr=03_basic_logs	2.0329e-3	2.0329e-3	99	Fri Apr 11, 14:56:31	2h 36m 52s
●	tfr=08_basic_logs	2.2988e-3	2.2988e-3	99	Fri Apr 11, 12:17:56	2h 39m 6s
○	tfr=1_basic_logs	2.6499e-3	2.6499e-3	99	Fri Apr 11, 09:37:07	2h 36m 53s

Figure 2: Teacher forching ratio legend

In the figure above, the green line represents a teacher forcing ratio (TFR) of 0.3, the red line corresponds to TFR = 0.8, and the blue line shows TFR = 1.0. According to the teacher forcing strategy described earlier, the first 10 epochs are trained using ground truth images rather than model-predicted ones. This typically results in more stable and efficient learning during this early phase. After the 10th epoch, training switches to using the model's own predictions as input, which is inherently less stable and leads to a temporary increase in loss. This behavior is clearly observed in the figure: after epoch 10, all three TFR settings experience a rise in loss and some instability for a few epochs before gradually decreasing again. Notably, models trained with lower TFR values show less fluctuation in loss, as they are introduced earlier to predicted inputs, which aligns with the core idea of teacher forcing—to help the model adapt to its own predictions over time.

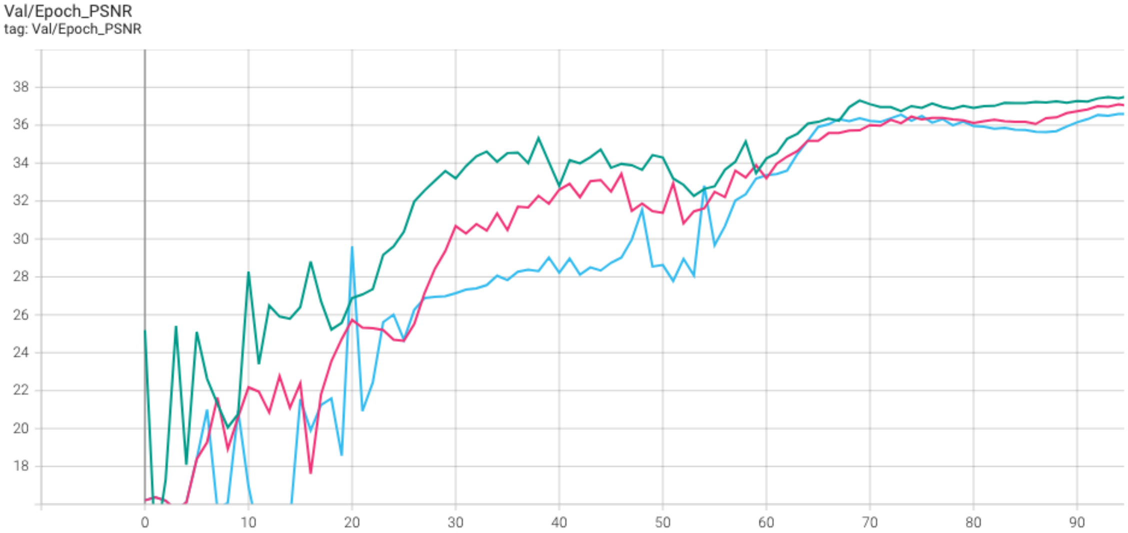


Figure 3: Teacher forching ratio validation PSNR comparison

It can be observed that models trained with a lower teacher forcing ratio (TFR) tend to have lower training loss and achieve higher validation PSNR across frames. A lower TFR exposes the model to its own predictions earlier, helping it adapt more effectively to inference conditions. This not only improves generalization but also accelerates convergence, resulting in faster training. Overall, using a lower TFR can lead to more efficient learning and better performance in video prediction tasks.

3.2 Plot the loss curve while training with different settings

3.2.1 Different KL annealing strategies

Below experiments are conducted with batch size = 12, learning rate = 5e-4, optimizer = AdamW, , number of epochs = 100, , tfr_sde = 10, tfr_d_step = 0.1, tfr = 1.0 , and different KL Divergence annealing strategies.

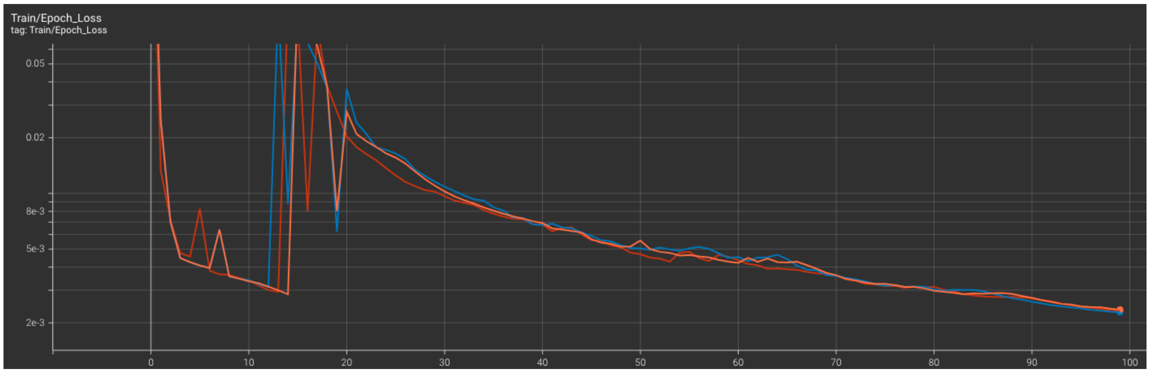
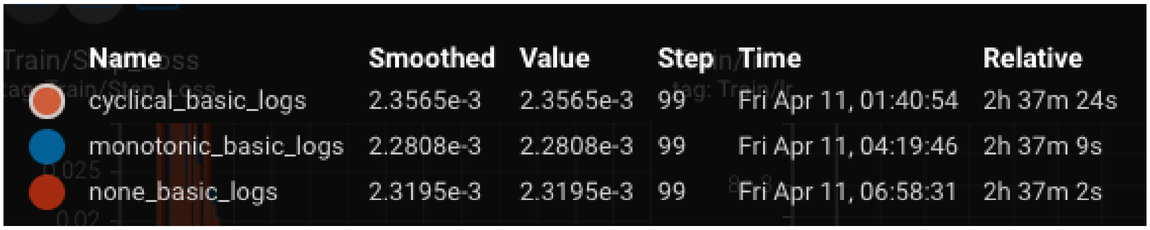


Figure 4: Different KL annealing strategies loss comparison



	Name	Smoothed	Value	Step	Time	Relative
●	cyclical_basic_logs	2.3565e-3	2.3565e-3	99	Fri Apr 11, 01:40:54	2h 37m 24s
●	monotonic_basic_logs	2.2808e-3	2.2808e-3	99	Fri Apr 11, 04:19:46	2h 37m 9s
●	none_basic_logs	2.3195e-3	2.3195e-3	99	Fri Apr 11, 06:58:31	2h 37m 2s

Figure 5: Different KL annealing strategies legend

In the figure above, the orange line represents the KL annealing strategy of cyclical, the blue line corresponds to the monotonic strategy, and the red line shows the strategy without annealing. Each behavior reflects the nature of its beta scheduling. The cyclical strategy causes the loss to increase and decrease periodically, as its beta value also rises and falls in cycles. The monotonic strategy results in a smoother and more gradual loss decrease, since its beta steadily increases over time. In contrast, the no-annealing strategy produces a loss curve that appears similar to monotonic in trend but exhibits greater instability, likely due to the constant high KL weight applied from the beginning of training.

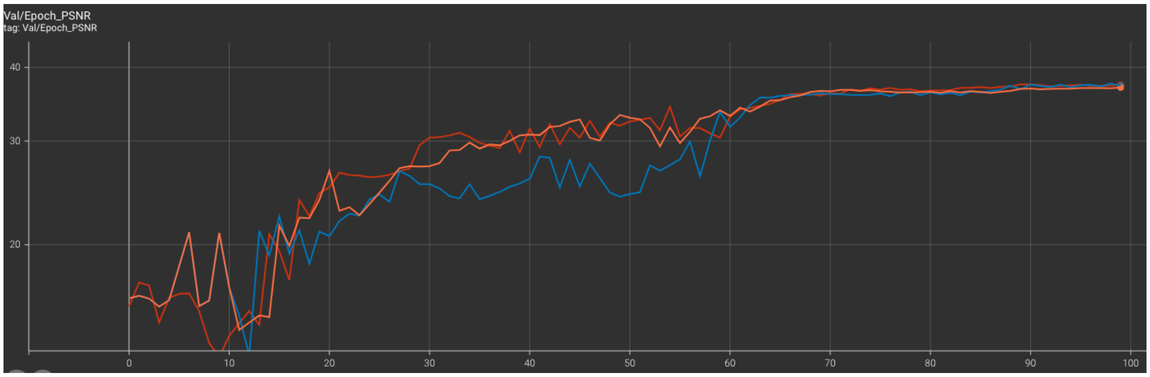


Figure 6: Different KL annealing strategies validation PSNR comparison

For the validation PSNR per frame, it can be observed that all three KL annealing strategies eventually achieve similar outcomes. The main differences appear during the early stages of training. Through multiple trials and experimentation, it was found that the cyclical strategy consistently produces slightly better results in terms of PSNR. This suggests that the periodic relaxation and reintroduction of KL regularization in the cyclical schedule may help the model explore the latent space more effectively and stabilize learning in early phases.

3.3 Plot the PSNR-per-frame diagram in the validation dataset

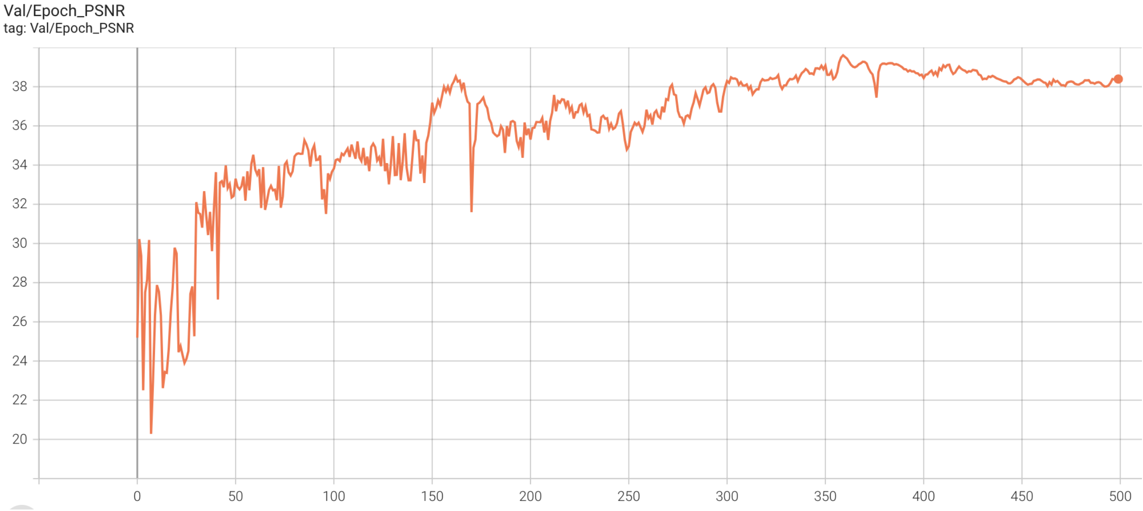


Figure 7: PSNR-per-frame in validation dataset

The above figure is PSNR-per-frame diagram in the validation dataset with best PSNR I discovered with hyperparamters:
batch size = 12, learning rate = 5e-4, AdamW optimizer, kl_anneal_type = Cyclical, number of

epochs = 500, tfr = 0.3, tfr_sde = 10 and tfr_d_step = 0.1
Can discover that PSNR grow gradually as epoch # grows.

3.4 Other training strategy analysis

3.4.1 Different optimizers

Below experiments are conducted with batch size = 12, learning rate = 5e-4(SGD's is 1e-2), KL Divergence annealing type = Cyclical, number of epochs = 500, tfr_sde = 10, tfr_d_step = 0.1, tfr = 0.3 and different types of optimizer.

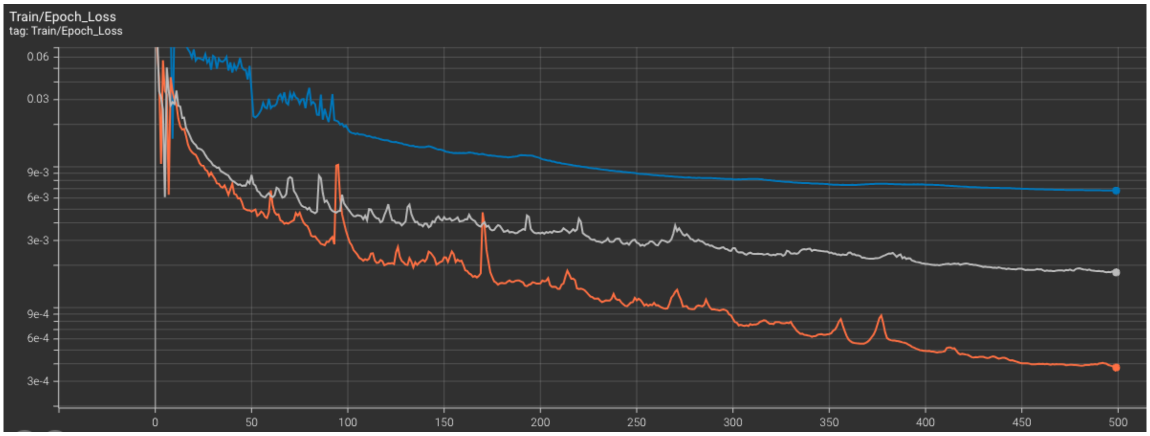
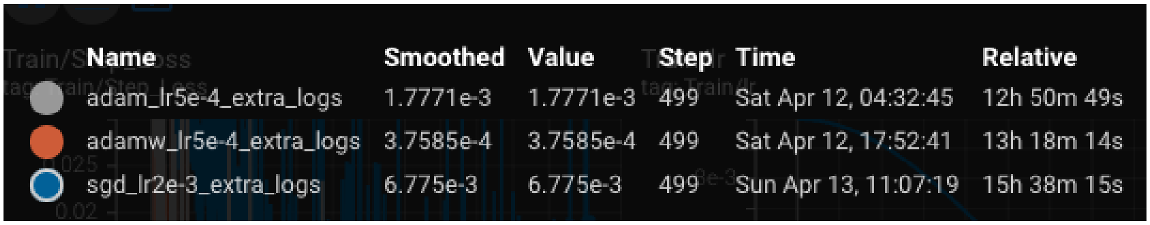


Figure 8: Different optimizer loss comparison



	Name	Smoothed	Value	Step	Time	Relative
●	adam_lr5e-4_extra_logs	1.7771e-3	1.7771e-3	499	Sat Apr 12, 04:32:45	12h 50m 49s
●	adamw_lr5e-4_extra_logs	3.7585e-4	3.7585e-4	499	Sat Apr 12, 17:52:41	13h 18m 14s
●	sgd_lr2e-3_extra_logs	6.775e-3	6.775e-3	499	Sun Apr 13, 11:07:19	15h 38m 15s

Figure 9: Different optimizer legend

In the figure above, the gray line represents the Adam optimizer, the orange line corresponds to AdamW, and the blue line shows the performance of SGD. From the figure, it can be observed that AdamW achieves the lowest loss, followed by Adam, while SGD performs the worst. For the task of VAE-based video prediction, AdamW performs best because it decouples weight decay from the gradient update, providing more effective regularization and improving stability during training. This is particularly beneficial for models with complex latent structures like VAEs. Adam also uses adaptive learning rates and momentum, making it more efficient than SGD, but it does not handle weight decay as effectively as AdamW. In contrast, SGD lacks both adaptive learning rates and built-in momentum by default, making it less suited for the high-dimensional, non-convex optimization landscape of video prediction models, resulting in slower convergence and higher loss.

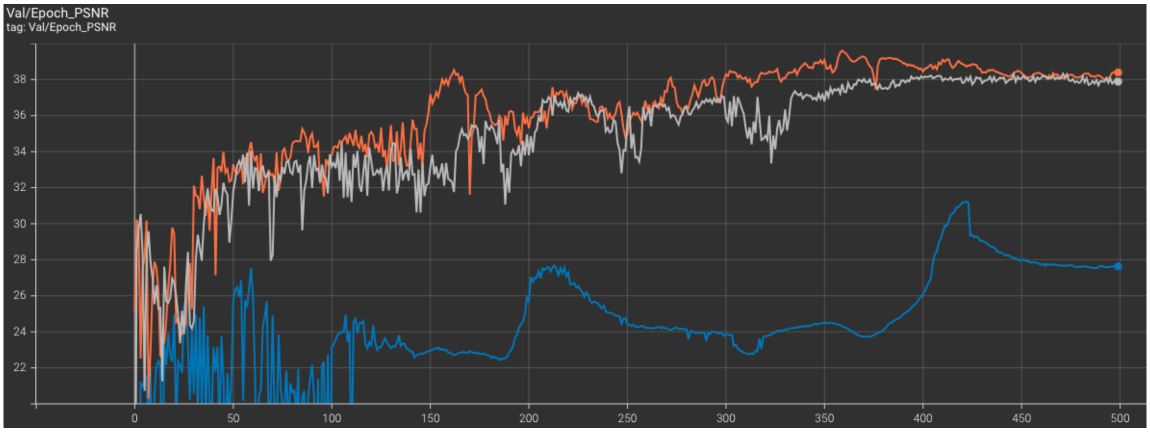


Figure 10: Different optimizer validation PSNR comparison

The outcome of validation psnr shows the same outcome as loss.

3.4.2 Clamping decoder output with Sigmoid

Below experiments are conducted with batch size = 12, learning rate = 5e-4(SGD's is 1e-2), KL Divergence annealing type = Cyclical, number of epochs = 500, tfr_sde = 10, tfr_d_step = 0.1, tfr = 0.3 and optimizer = AdamW. The difference is that I use sigmoid function to clamp decoder fusion output instead of torch build in function as I discover that it will generate nan which will cause VAE failed.

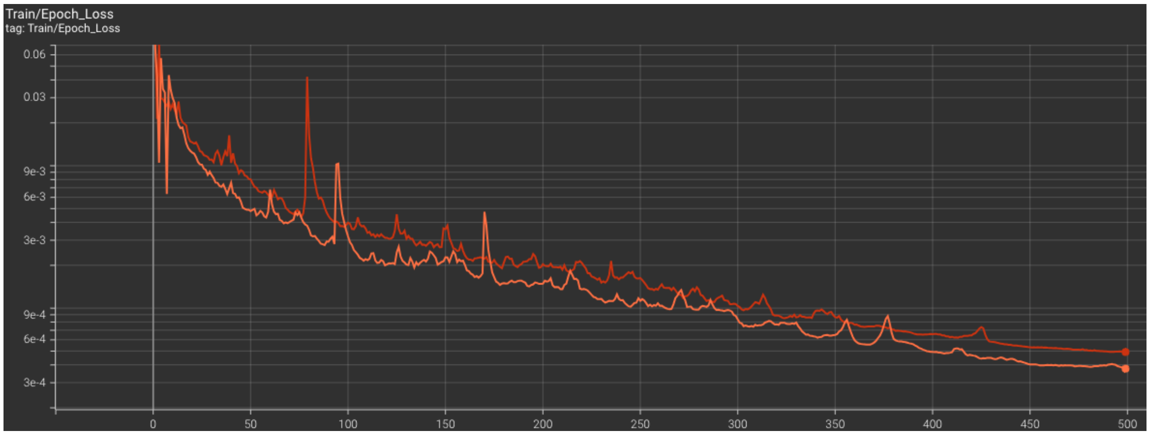
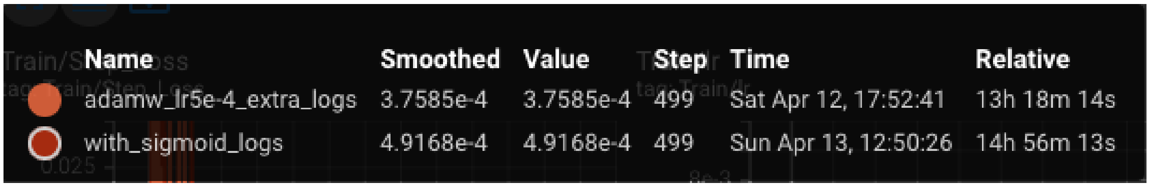


Figure 11: Loss comparison of w & w/o sigmoid



	Name	Smoothed	Value	Step	Time	Relative
●	adamw_lr5e-4_extra_logs	3.7585e-4	3.7585e-4	499	Sat Apr 12, 17:52:41	13h 18m 14s
●	with_sigmoid_logs	4.9168e-4	4.9168e-4	499	Sun Apr 13, 12:50:26	14h 56m 13s

Figure 12: w & w/o sigmoid legend

In the figure above, the orange line represents the model without using a sigmoid function for clamping, while the red line corresponds to the model with sigmoid applied. It was initially expected that applying a sigmoid function would result in a smoother clamping effect compared to using a built-in function like torch.clamp, which enforces hard boundaries by directly assigning values below 0 to 0 and above 1 to 1. However, the loss curve shows that the use of sigmoid did not lead to improved performance. This is likely because, although sigmoid smoothly bounds the output between 0 and 1, it can also cause gradient saturation—when the input is far from zero, the gradient becomes very small, slowing down learning. In contrast, hard clamping with torch.clamp allows for stronger gradient flow in the central region of the output, which may be more beneficial for learning in pixel prediction tasks like video frame reconstruction.

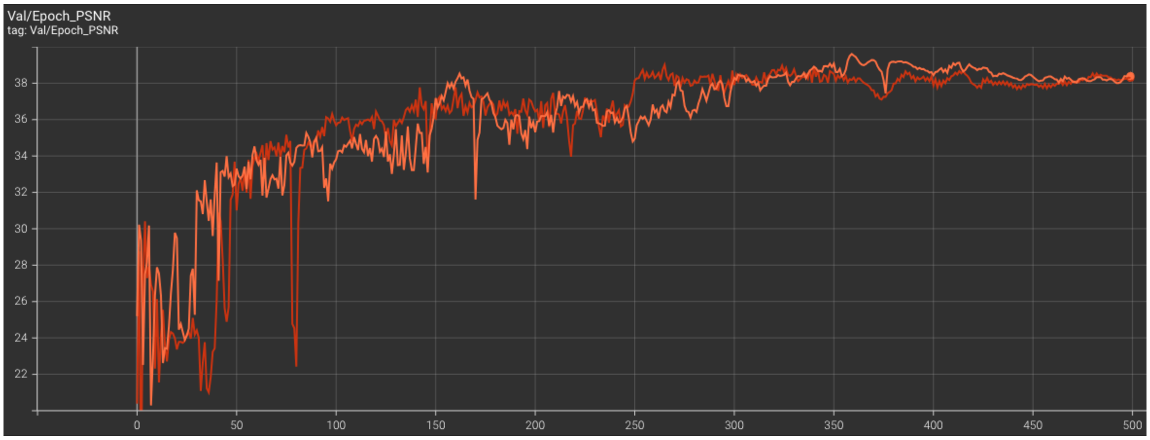


Figure 13: w & w/o sigmoid validation PSNR comparison